fluid

Bacco, Giacomo

Citing

Please cite this work if you have used it in your project or in your program.

Giacomo Bacco. *fluid: Free Fluid Flux-Barriers Rotor for Synchronous Reluctance Motor Drawing*. 2018. URL: `https://github.com/gbacco5/fluid`

Contents

Goal

Provide a ready-to-use fully parametric drawing of the Synchronous Reluctance Rotor with fluid flux-barriers.

The scope of this project is the computation of the flux-barriers points. The drawing scripts are for demonstration purposes only.

Requirements

Matlab or Octave or Python (NumPy + SciPy) to compute the points. The points calculation is general, so it could be implemented in any language, but I chose Matlab/Octave because it is my standard interface with FEMM software.

If you do not use FEMM, you can still use the calculation part and make a porting for your CAD engine or FEA software. If you do so, consider contributing to the project adding your interface scripting.

1 Files needed

For Matlab/Octave, the files needed for the computation are
`calc_fluid_barrier.m`,
`GetFSolveOptions.m` and
`isOctave.m`
while for Python it is
`fluid_functions.py`.

Contacts

You can contact me at giacomo.bacco@phd.unipd.it.

If you find a bug, consider opening an issue at `https://github.com/gbacco5/fluid/issues`

Last update: November 24, 2018

Notice:

2 How to use

Open the file `fluid` and run it.

Change the machine data in the data section. All the variables have a comment next to them.

There are some "hidden" options which should be explained.

1. you can provide personal flux-barrier angles, or let the program compute them. This means that you have to always provide the flux-barrier thicknesses and flux-carrier widths. Optionally, you could also provide the electrical flux-barrier angles.

```
rotor.barrier_angles_el = [14,26,38]*2;
```

2. if you let the program compute them, they will result the average of the final points C' and D' at the rotor periphery. Alternatively, you can define some weights which determine how close E should be to C' or D'.

```
rotor.barrier_end_wf = [20,50,80]/100;
```

3. by default, the flux-barrier-end is round, so the code solves an additional system to determine the correct locations of the fillet points. You can skip such system declaring

```
rotor.barrier_end = 'rect';
```

   and so selecting "rectangular" flux-barrier-end.

4. the inner radial iron ribs are optional, but you are free to provide different widths for every flux-barrier.

```
rotor.wrib = [1,2,4]*mm;
```

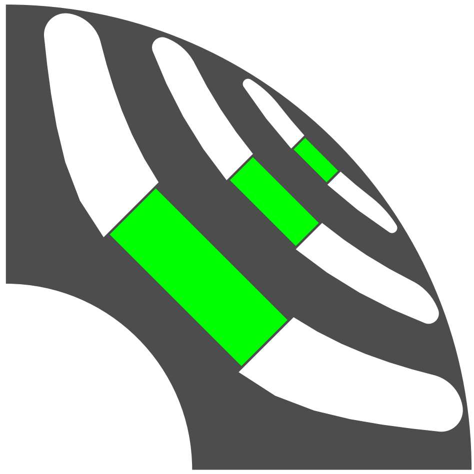
5. if you also input magnet widths, the rib is automatically enlarged to accommodate the magnet, similarly to an IPM (Interior Permanent Magnet) machine.

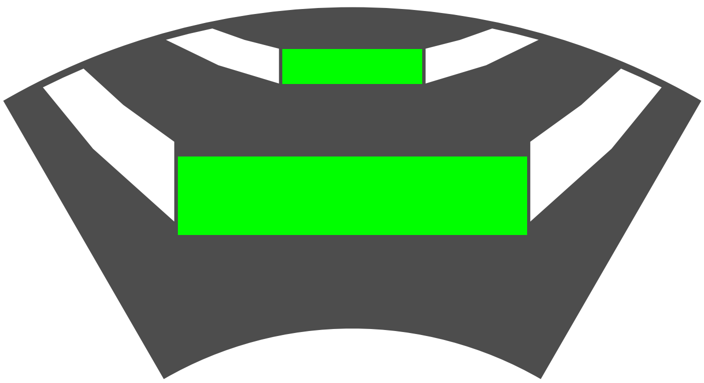
```
rotor.wm = [10,20,40]*mm;
```

   In this case, the output structure `barrier` also contains the location of the magnet base center point.

3 Examples

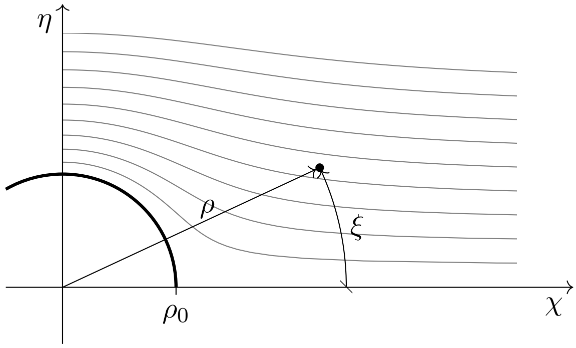
Here are some finished examples based on the output. The drawings are for demonstration purposes only.





4 Theory

4.1 Flow past a cylinder



Let ρ_0 be the radius of the cylinder, ρ, ξ the polar coordinate system in use. One possible solution of this problem have these potential and streamline functions:

$$\phi(\rho, \xi) = \left(\rho + \frac{\rho_0^2}{\rho}\right) \cos\xi \tag{1}$$

$$\psi(\rho, \xi) = \left(\rho - \frac{\rho_0^2}{\rho}\right) \sin\xi \tag{2}$$

Although these equations are deeply coupled, the radius ρ and the phase ξ can be obtained as a function of the other quantities. For our purposes, we use ψ.

$$\rho(\psi, \xi) = \frac{\psi + \sqrt{\psi^2 + 4\rho_0^2 \sin^2\xi}}{2\sin\xi} \tag{3}$$

$$\xi(\psi, \rho) = \arcsin\left(\frac{\rho\,\psi}{\rho^2 - \rho_0^2}\right) \tag{4}$$

The velocity field can also be derived through

$$\begin{aligned} v_\rho(\rho, \xi) &= \frac{\partial\phi}{\partial\rho} = \left(1 - \frac{\rho_0^2}{\rho^2}\right)\cos\xi \\ v_\xi(\rho, \xi) &= \frac{1}{\rho}\frac{\partial\phi}{\partial\xi} = -\left(1 + \frac{\rho_0^2}{\rho^2}\right)\sin\xi \end{aligned} \tag{5}$$

4.2 Conformal mapping

From the reference plane, which is equivalent to a two-pole machine, we use a complex map to obtain the quantities in the actual plane. Let p be the number of pole pairs. Then:

$$
\begin{aligned}
\zeta &\xrightarrow{\mathcal{M}} z = \sqrt[p]{\zeta} \\
\rho\, \mathrm{e}^{\mathrm{j}\xi} &\xrightarrow{\mathcal{M}} r\, \mathrm{e}^{\mathrm{j}\vartheta} = \sqrt[p]{\rho}\, \mathrm{e}^{\mathrm{j}\xi/p} \\
\chi + \mathrm{j}\eta &\xrightarrow{\mathcal{M}} x + \mathrm{j}y
\end{aligned} \tag{6}
$$

It is easy to find the inverse map:

$$
\mathcal{M}\colon \sqrt[p]{\cdot} \qquad \mathcal{M}^{-1}\colon (.)^p \tag{7}
$$

In the transformed plane, the velocities have a different expression:

$$
\begin{aligned}
v_r(r,\vartheta) &= p\left(r^{p-1} - \frac{R_0^{2p}}{r^{p+1}}\right)\cos p\vartheta \\
v_\vartheta(r,\vartheta) &= -p\left(r^{p-1} + \frac{R_0^{2p}}{r^{p+1}}\right)\sin p\vartheta
\end{aligned} \tag{8}
$$

This vector field is tangent to the streamlines in every point in the transformed plane. In order to work with this field in x, y coordinates, we need a rotational map:

$$
\begin{aligned}
v_x(r,\vartheta) &= v_r \cos\vartheta - v_\vartheta \sin\vartheta \\
v_y(r,\vartheta) &= v_r \sin\vartheta + v_\vartheta \cos\vartheta
\end{aligned} \tag{9}
$$

4.3 Computation of flux-barrier base points

Refer to Figure 1 for the points naming scheme. Keep in mind that A' is not simply the projection of A onto the q-axis, but it represents the original starting point for the barrier sideline, so it lies on the flux-barrier streamline. The same is true for points B', B, C', C, and D', D.

Let the flux-barrier and flux-carrier thicknesses and widths be given. Then the base points for the flux-barriers can be computed easily. Let D_r

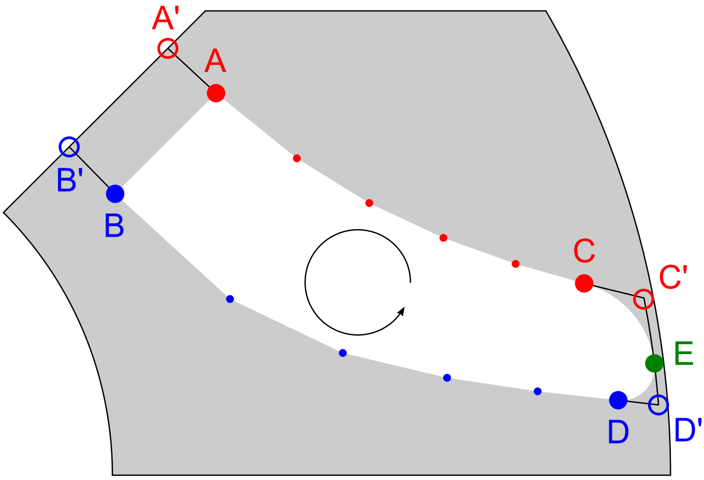


Figure 1: Flux-barrier base points description.

be the rotor outer diameter, $w_{\mathrm{rib,t}}$ the tangential iron rib width, $w_{\mathrm{c},k}$ the k-th flux-carrier width, and $t_{\mathrm{b},k}$ the k-th flux-barrier thickness.[1] Then

$$\begin{aligned}
R_{\mathrm{rib}} &= \frac{D_{\mathrm{r}}}{2} - w_{\mathrm{rib,t}} \\
R_{\mathsf{A}'_1} &= R_{\mathrm{rib}} - w_{\mathrm{c},1} \\
R_{\mathsf{B}'_1} &= R_{\mathsf{A}'_1} - t_{\mathrm{b},1} \\
&\vdots
\end{aligned} \tag{10}$$

where R represents the radius from the origin. So, in general:

$$\begin{aligned}
R_{\mathsf{A}'_\mathsf{k}} &= R_{\mathsf{B}'_{\mathsf{k-1}}} - w_{\mathrm{c},k} \\
R_{\mathsf{B}'_\mathsf{k}} &= R_{\mathsf{A}'_{\mathsf{k-1}}} - t_{\mathrm{b},k}
\end{aligned} \tag{11}$$

with the exception $R_{\mathsf{B}'_0} = R_{\mathrm{rib}}$.

[1] You may wonder why the main dimensions of the flux-carrier and flux-barrier differ in the name (width versus thickness). This is due to a choice of mine, because I prefer to refer to width when the flux flows perpendicularly to the dimension, and to thickness when it flows in parallel.

Now we know both the radii and the angle – always $\pi/(2p)$ – of the flux-barrier internal points. So we can compute their respective streamline value.

4.3.1 Magnet insertion

$$w_{\mathrm{rib},k} \leftarrow w_{\mathrm{rib},k} + w_{\mathrm{m},k}$$

where $w_{\mathrm{m},k}$ is the k-th magnet width.

4.3.2 Central base points

We refer to points A and B. If the rib width is zero $\mathsf{A} \equiv \mathsf{A}'$ and $\mathsf{B} \equiv \mathsf{B}'$.

The line describing the q-axis is

$$\begin{aligned} y &= mx + q \\ m &= \tan\frac{\pi}{2p} \\ q &= \frac{w_{\mathrm{rib}}}{2\cos\frac{\pi}{2p}} \end{aligned} \tag{12}$$

$$\begin{cases} y_{\mathsf{A}} - m x_{\mathsf{A}} - q = 0 \\ x_{\mathsf{A}} - r_{\mathsf{A}}(\psi_{\mathsf{A}'}, \vartheta_{\mathsf{A}}) \cos\vartheta_{\mathsf{A}} = 0 \\ y_{\mathsf{A}} - r_{\mathsf{A}}(\psi_{\mathsf{A}'}, \vartheta_{\mathsf{A}}) \sin\vartheta_{\mathsf{A}} = 0 \end{cases} \tag{13}$$

where ϑ_{A} is used as the third degree of freedom and r_{A} is then a function of it. The solution of such system can be determined solving the single equation

$$r_{\mathsf{A}}(\psi_{\mathsf{A}'}, \vartheta_{\mathsf{A}})\big(\sin\vartheta_{\mathsf{A}} - m\cos\vartheta_{\mathsf{A}}\big) - q = 0 \tag{14}$$

in the unknown ϑ_{A}. The function $r(\psi, \vartheta)$ is simply

$$r(\psi, \vartheta) = \sqrt[p]{\rho(\psi, \vartheta/p))}$$

The same equation can be written for point B with the proper substitution and repeated for all the flux-barriers.

4.4 Outer base points

We refer to points C, D, and E. If the flux-barrier angle, α_{b}, is given, then

$$\begin{aligned} x_{\mathsf{E}} &= R_{\mathrm{rib}} \cos(\tfrac{\pi}{2p} - \alpha_{\mathrm{b}}) \\ y_{\mathsf{E}} &= R_{\mathrm{rib}} \sin(\tfrac{\pi}{2p} - \alpha_{\mathrm{b}}) \end{aligned} \tag{15}$$

Points C and D results from the connection of the flux-barrier sidelines and point E. This connection should be as smooth as possible in order to avoid dangerous mechanical stress concentrations. We are going to use circular arcs to make this connection. So we impose the tangency between the flux-barrier sideline and the arc, between the arc and the radius through point E. The tangent to the sideline can be obtained through the velocity field described above.

Then we want point C to lay on the flux-barrier sideline. These conditions represent a nonlinear system of 4 equations, in 6 unknowns. So we need two more equations, which are that points C and E belong to the fillet circle with radius R.

$$\begin{cases} x_{\mathsf{C}} - r_{\mathsf{C}}(\psi_{\mathsf{C}}, \vartheta_{\mathsf{C}}) \cos\vartheta_{\mathsf{C}} = 0 \\ y_{\mathsf{C}} - r_{\mathsf{C}}(\psi_{\mathsf{C}}, \vartheta_{\mathsf{C}}) \sin\vartheta_{\mathsf{C}} = 0 \\ (x_{\mathsf{C}} - x_{\mathsf{O_C}})^2 + (y_{\mathsf{C}} - y_{\mathsf{O_C}})^2 - R_{\mathsf{EC}}^2 = 0 \\ (x_{\mathsf{E}} - x_{\mathsf{O_C}})^2 + (y_{\mathsf{E}} - y_{\mathsf{O_C}})^2 - R_{\mathsf{EC}}^2 = 0 \\ (x_{\mathsf{O_C}} - x_{\mathsf{E}})y_{\mathsf{E}} - (y_{\mathsf{O_C}} - y_{\mathsf{E}})x_{\mathsf{E}} = 0 \\ (x_{\mathsf{O_C}} - x_{\mathsf{C}})v_x(r_{\mathsf{C}}, \vartheta_{\mathsf{C}}) + (y_{\mathsf{O_C}} - y_{\mathsf{C}})v_y(r_{\mathsf{C}}, \vartheta_{\mathsf{C}}) = 0 \end{cases} \tag{16}$$

The very same system can be written and solved for point D.

4.4.1 Choice of initial position

For the good convergence of the nonlinear system, we have to choose a proper initial position for the points of interest, namely C and $\mathsf{O_C}$ for the top part of the flux-barrier.

Since point C should be close to E and C', a good initial guess could be

$$x_{\mathsf{C}^{(0)}} = \frac{x_{\mathsf{E}} + x_{\mathsf{C}'}}{2}\,, \qquad y_{\mathsf{C}^{(0)}} = \frac{y_{\mathsf{E}} + y_{\mathsf{C}'}}{2} \tag{17}$$

A slightly better guess shifts the points a bit to the left, in this way:

$$x_{\mathsf{C}^{(0)}} = \frac{x_{\mathsf{E}} + x_{\mathsf{C}'} + 0.1 x_{\mathsf{A}}}{2.1}\,, \qquad y_{\mathsf{C}^{(0)}} = \frac{y_{\mathsf{E}} + y_{\mathsf{C}'}}{2} \tag{18}$$

On the other hand, point $\mathsf{O_C}$ lies on one edge of the triangle of vertices $\mathsf{E}, \mathsf{C}, \mathsf{O}$, where O represents the origin and where we are going to use $\mathsf{C}^{(0)}$ instead of C because it is still unknown. Then:

$$x_{\mathsf{O_C}}^{(0)} = \frac{x_{\mathsf{E}} + x_{\mathsf{C}^{(0)}} + 0}{3}\,, \qquad y_{\mathsf{O_C}}^{(0)} = \frac{y_{\mathsf{E}} + y_{\mathsf{C}^{(0)}} + 0}{3} \tag{19}$$

Similar considerations can be made for point D, with slight changes:

$$x_{\mathsf{D}^{(0)}} = \frac{x_{\mathsf{E}} + x_{\mathsf{D}'}}{2}\,, \qquad y_{\mathsf{D}^{(0)}} = \frac{y_{\mathsf{E}} + y_{\mathsf{D}'}}{2} \tag{20}$$

$$x_{\mathsf{O_D}}^{(0)} = \frac{x_{\mathsf{E}} + x_{\mathsf{D}^{(0)}} + x_{\mathsf{C}}}{3}\,, \qquad y_{\mathsf{O_D}}^{(0)} = \frac{y_{\mathsf{E}} + y_{\mathsf{D}^{(0)}} + x_{\mathsf{C}}}{3} \tag{21}$$

Notice that here we use point C which has already been found.

4.5 Flux-barrier sideline points

Consider the top flux-barrier sideline, so the one going from point A to point C. We want to create such sideline using a predetermined number of steps, N_{step}. From now on, let us call this number N, and N_k for the k-th flux-barrier.

One of the best way to distribute the points along the streamline is to use the potential function, ϕ, defined in Equation 1. We start computing the potential for points A and C:

$$\phi_{\mathsf{A}} = \phi(\rho_{\mathsf{A}}, \xi_{\mathsf{A}})$$
$$\phi_{\mathsf{C}} = \phi(\rho_{\mathsf{C}}, \xi_{\mathsf{C}})$$

Then, we want to find $N-1$ points along the streamline between points A and C with a uniform distribution of the potential function. We define

$$\Delta\phi_{\mathsf{AC}} = \frac{\phi_{\mathsf{C}} - \phi_{\mathsf{A}}}{N}$$

So we can compute the potentials we are looking for

$$\phi_i = \phi_{\mathsf{A}} + i\Delta\phi_{\mathsf{AC}}\,, \quad i = 1, \ldots, N-1$$

and finally the location of the point with this potential value and the streamline function value required to lie on the flux-barrier sideline. This translates to the following system of equations:

$$\begin{cases} \psi_{\mathsf{AC}} - \psi(\rho, \xi) = 0 \\ \phi_i - \phi(\rho, \xi) = 0 \end{cases} \tag{22}$$

The system is well-defined because there are two unknowns and two independent equations. This system must be solved for every flux-barrier sideline point, for the two sides, and for every flux-barrier.[2]

4.6 Output

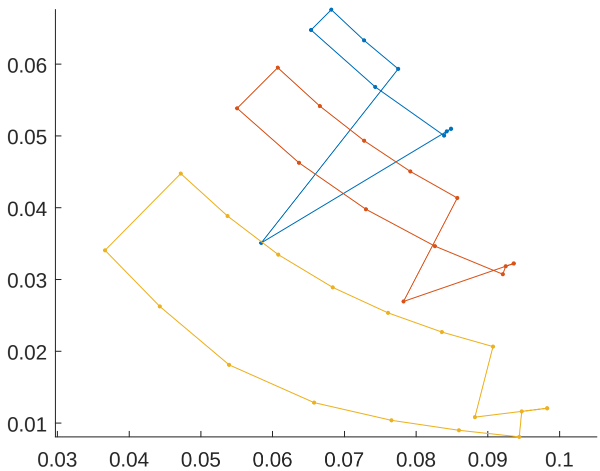
The output of the computation function in Matlab/Octave is one vector of structures (`barrier(:)`) which contains at least two fields (`X` and `Y`). The X vector is made in this way:

$$X = \begin{bmatrix} x_{\mathsf{E}} & x_{\mathsf{O_C}} & x_{\mathsf{C}} & x_{\mathsf{AC}_{N_{\text{step}}-1}} & \cdots & x_{\mathsf{AC}_1} & x_{\mathsf{A}} & \searrow \\ x_{\mathsf{B}} & x_{\mathsf{BD}_1} & \cdots & x_{\mathsf{BD}_{N_{\text{step}}-1}} & x_{\mathsf{D}} & x_{\mathsf{O_D}} & x_{\mathsf{E}} & \end{bmatrix}^{\mathrm{T}}$$

and similarly the Y vector. So the points are ordered starting from the point E and then moving counter-clockwise until E is reached again.

[2]In Matlab/Octave, the "for every flux-barrier sideline point" loop has been vectorized, while the two sides has been manually split.

4.7 Example of Matlab/Octave plot



Here is an example of a Matlab/Octave output plot. The V-shaped lines represent the radii of the fillet arcs, which were not worth to be shown in Matlab/Octave.

5 Matlab Code

5.1 Main file

```
% FLUID
% Free Fluid Flux-Barriers Rotor for Synchronous Reluctance Motor Drawing
%
% Bacco, Giacomo 2018

clear all; close all; clc;
addpath('draw','tools');

%% DATA
rotor.p = 2; % number of pole pairs
mm = 1e-3; % millimeters
rotor.De = 200*mm; % [m], rotor outer diameter

rotor.Nb = 3; % number of flux-barriers
rotor.tb = [4 8 15]*mm; % flux-barrier thicknesses
rotor.wc = [3 7 12 10]*mm; % flux-carrier widths
rotor.Nstep = 3*[2, 4, 6]; % number of steps to draw the flux-barrier side
rotor.wrib_t = 1*mm; % [m], tangential iron rib width

% you can input flux-barrier angles or let the program compute them
% rotor.barrier_angles_el = [14,26,38]*2; % [deg], electrical flux-barrier angles

% or you can also input a factor to reach the top or the bottom of each barrier
% (do not exceed 100%)
% rotor.barrier_end_wf = [20,50,80]/100; % flux-barrier-end weight factors

% rotor.barrier_end = 'rect'; % choose 'rect' or comment

% you can define the rib width or comment
rotor.wrib = [0,1,1]*mm; % [m], radial iron rib widths
% You can define the magnet width or comment
```

```
% rotor.wm = [10,20,40]*mm;

%% barrier points computation
barrier = calc_fluid_barrier(rotor);

%% simple matlab plot
figure
hold all
axis equal
for bkk = 1:rotor.Nb
  plot(barrier(bkk).X, barrier(bkk).Y, '.-')
end
if isfield(rotor,'wm')
  RM = [barrier(:).Rm];
  thM = pi/2/rotor.p;
  Xm = RM.*cos(thM);
  Ym = RM.*sin(thM);
  plot(Xm, Ym, 'ko')
end
axis auto

%% FEMM drawing
try
  openfemm(1)
  newdocument(0);

  draw_fluid_barrier(barrier);

catch
  disp('FEMM not available.');

end
```

5.2 Calc fluid barrier

```matlab
function barrier = calc_fluid_barrier(r)
% CALC_FLUID_BARRIER computes the flux-barrier points along the streamline
% function.

%% DATA
global deb

Dr = r.De; % [m], rotor outer diameter
ScalingFactor = 1/( 10^(round(log10(Dr))) );
% ScalingFactor = 1;
Dr = Dr*ScalingFactor;

p = r.p; % number of pole pairs
Nb = r.Nb; % number of flux-barriers
tb = r.tb*ScalingFactor; % flux-barrier widths
wc = r.wc*ScalingFactor; % flux-carrier widths
Nstep = r.Nstep; % number of steps to draw the flux-barrier side

wrib_t = r.wrib_t*ScalingFactor; % [m], tangential iron rib width

if isfield(r,'barrier_angles_el')
  barrier_angles_el = r.barrier_angles_el; % [deg], electrical flux-barrier angles
  AutoBarrierEndCalc = 0;
else
  barrier_angles_el = zeros(1,Nb);
  AutoBarrierEndCalc = 1;
  if isfield(r,'barrier_end_wf')
    wf = r.barrier_end_wf;
  else
    wf = 0.5*ones(1,Nb);
  end
end

if isfield(r,'wm')
  wm = r.wm*ScalingFactor;
```

```
else
  wm = 0;
end
if isfield(r,'wrib')
  wrib = r.wrib*ScalingFactor + wm; % [m], radial iron rib widths
else
  wrib = zeros(1,Nb) + wm;
end

Dend = Dr - 2*wrib_t; % [m], flux-barrier end diameter
Dsh = Dend - 2*( sum(tb) + sum(wc) ); % [m], shaft diameter
R0 = Dsh/2; % [m], shaft radius
barrier_angles = barrier_angles_el/p; % [deg], flux-barrier angles
if isfield(r,'barrier_end')
  barrier_end = r.barrier_end;
else
  barrier_end = '';
end

%% IMPLICIT FUNCTIONS
% definition of fluid past a cylinder functions
psi_fluid = @(rho,xi,rho0) (rho.^2 - rho0^2)./rho.*sin(xi);
phi_fluid = @(rho,xi,rho0) (rho.^2 + rho0^2)./rho.*cos(xi);
xi_fluid  = @(psi,rho,rho0) asin(psi.*rho./(rho.^2 - rho0^2));
rho_fluid = @(psi,xi,rho0) ( psi + sqrt(psi.^2 + 4*sin(xi).^2*rho0^2) )./(2*sin(xi));

r_map = @(rho) rho.^(1/p);
th_map = @(xi) xi./p;
rho_map = @(r) r.^p;
xi_map = @(th) th.*p;

vr = @(r,th,R0) p*(r.^(p-1) - R0^(2*p)./r.^(p+1)).*cos(p*th);
vt = @(r,th,R0) -p*(r.^(p-1) + R0^(2*p)./r.^(p+1)).*sin(p*th);
vx = @(vr_v,vth_v,th) vr_v.*cos(th) - vth_v.*sin(th);
vy = @(vr_v,vth_v,th) vr_v.*sin(th) + vth_v.*cos(th);

%% Precomputations
rho0 = rho_map(R0);
```

```
%% Central base points
RAprime = Dend(1)/2 - [0, cumsum( tb(1:end-1))] - cumsum(wc(1:end-1)); % top
RBprime = RAprime - tb; % bottom
te_qAxis = pi/(2*p); % q-axis angle in rotor reference frame

% get A' and B' considering rib and magnet widths
mCentral = tan(te_qAxis); % slope
qCentral = repmat( -wrib/2/cos(te_qAxis), 1, 2); % intercept

psiCentralPtA = psi_fluid(rho_map(RAprime), xi_map(te_qAxis), rho0);
psiCentralPtB = psi_fluid(rho_map(RBprime), xi_map(te_qAxis), rho0);
psiCentralPt = [psiCentralPtA, psiCentralPtB];
psiA = psiCentralPtA;
psiB = psiCentralPtB;

CentralPt_Eq = @(th) ...
  r_map( rho_fluid(psiCentralPt, xi_map(th), rho0) ).*...
  ( sin(th) - mCentral*cos(th) ) - qCentral;

if deb == 1
  options.Display = 'iter'; % turn off folve display
else
  options.Display = 'off'; % turn off folve display
end
options.Algorithm = 'levenberg-marquardt'; % non-square systems
options.FunctionTolerance = 1*eps;
options.TolFun = options.FunctionTolerance;
options.StepTolerance = 1e4*eps;
options.TolX = options.StepTolerance;
% I thought the new Matlab syntax for fsolve was options.FunctionTolerance,
% I was wrong.

X0 = repmat(te_qAxis,1,2*Nb);
options = GetFSolveOptions(options);
```

```
teAB = fsolve(CentralPt_Eq, X0, options);
teA = teAB(1:Nb);
teB = teAB(Nb+1:end);
RA = r_map( rho_fluid(psiA, xi_map(teA), rho0) );
RB = r_map( rho_fluid(psiB, xi_map(teB), rho0) );

% central base points
xA = real( RA.*exp(1j*teA) );
yA = imag( RA.*exp(1j*teA) );
xB = real( RB.*exp(1j*teB) );
yB = imag( RB.*exp(1j*teB) );

% magnet central base point radius computation
RAsecond = RA.*cos(te_qAxis - teA);
RBsecond = RB.*cos(te_qAxis - teB);

Rmag = (RAprime + RAsecond + RBprime + RBsecond)/4;

%% Outer base points C,D preparation
RCprime = Dend/2;
teCprime = th_map( xi_fluid(psiA, rho_map(RCprime), rho0) );
xCprime = Dend/2.*cos(teCprime);
yCprime = Dend/2.*sin(teCprime);

RDprime = Dend/2;
teDprime = th_map( xi_fluid(psiB, rho_map(RDprime), rho0) );
xDprime = Dend/2.*cos(teDprime);
yDprime = Dend/2.*sin(teDprime);

if AutoBarrierEndCalc
  teE = ( teCprime.*(1 - wf) + teDprime.*wf );
  aphE = pi/2/p - teE;
  barrier_angles = 180/pi*aphE;
  barrier_angles_el = p*barrier_angles;
else
  aphE = barrier_angles*pi/180;
  teE = pi/2/p - aphE;
end
xE = Dend/2.*cos(teE);
yE = Dend/2.*sin(teE);
```

```
%% Outer base points C (top)
if strcmp(barrier_end, 'rect')
  RC = RCprime;
  teC = teCprime;
  xC = xCprime;
  yC = yCprime;
  xOC = xC;
  yOC = yC;

else
  options.Algorithm = 'trust-region-dogleg'; % non-square
    systems

  BarrierEndSystem = @(th,xd,yd,xo,yo,R) ...
    [xd - r_map(rho_fluid(psiA', p*th, rho0)).*cos( th )
    yd - r_map(rho_fluid(psiA', p*th, rho0)).*sin( th )
    (xd - xo).^2 + (yd - yo).^2 - R.^2
    (xE' - xo).^2 + (yE' - yo).^2 - R.^2
    (xo - xd).*vx( vr( r_map(rho_fluid(psiA', p*th, rho0)),th,R0
    ), vt( r_map(rho_fluid(psiA', p*th, rho0)) ,th,R0 ), th) +
    (yo - yd).*vy(  vr( r_map(rho_fluid(psiA', p*th, rho0)),th,
    R0 ), vt( r_map(rho_fluid(psiA', p*th, rho0)) ,th,R0 ), th)
    (xo - xE').*yE' - (yo - yE').*xE'
    %   th - xi_fluid((rho_fluid(p*th, psiA', rho0)),
    psiA', rho0)/p % serve?
    ];

%   X0 = [ aph_b, 0, 0, 0, 0, 0]; % 1st try
%   X0 = [ 1.5*teE', 0.9*xE', 0.9*yE', 0.8*xE', 0.8*yE',
    0.25*xE']; % 2nd try
% best try
  xC0 = ( xE + xCprime + 0.1*xA )/(2 + 0.1);
  yC0 = ( yE + yCprime )/2;
  thC0 = atan(yC0./xC0);
  xOC0 = ( xE + xC0 + 0 )/3;
  yOC0 = ( yE + yC0 + 0 )/3;
  RCOCE0 = sqrt( (xOC0 - xE).^2 + (yOC0 - yE).^2 );

  X0 = [ thC0', xC0', yC0', xOC0', yOC0', RCOCE0'];
  X = fsolve( @(x) BarrierEndSystem( x(:,1),x(:,2),x(:,3),x(:,4)
    ,x(:,5),x(:,6) ), X0, options);
```

```
  xOC = X(:,4)';
  yOC = X(:,5)';
  xC = X(:,2)';
  yC = X(:,3)';
  RC = hypot(xC, yC);
  teC = atan2(yC, xC);
end

%% Outer base points D (bottom)
if strcmp(barrier_end, 'rect')
  RD = RDprime;
  teD = teDprime;
  xD = xDprime;
  yD = yDprime;
  xOD = xD;
  yOD = yD;

else
  options.Algorithm = 'levenberg-marquardt'; % non-square
    systems

  BarrierEndSystem = @(th,xd,yd,xo,yo,R) ...
    [xd - r_map(rho_fluid(psiB', p*th, rho0)).*cos( th )
    yd - r_map(rho_fluid(psiB', p*th, rho0)).*sin( th )
    (xd - xo).^2 + (yd - yo).^2 - R.^2
    (xE' - xo).^2 + (yE' - yo).^2 - R.^2
    (xo - xd).*vx( vr( r_map(rho_fluid(psiB', p*th, rho0)),th,R0
     ), vt( r_map(rho_fluid(psiB', p*th, rho0)) ,th,R0 ), th) +
    (yo - yd).*vy(  vr( r_map(rho_fluid(psiB', p*th, rho0)),th,
    R0 ), vt( r_map(rho_fluid(psiB', p*th, rho0)) ,th,R0 ), th)
    (xo - xE').*yE' - (yo - yE').*xE'
    %   th - xi_fluid((rho_fluid(p*th, psi_d, rho0)),
    psi_d, rho0)/p % serve?
    ];

%   X0 = [ 0.8*teE', 0.8*xE', 0.8*yE', xE'*.9, yE'*.9, xE
    '*.2]; % 1st try
% best try
  xD0 = ( xE + xDprime )/2;
  yD0 = ( yE + yDprime )/2;
```

```
  thD0 = atan(yD0./xD0);
  xOD0 = ( xE + xD0 + xC )/3;
  yOD0 = ( yE + yD0 + yC )/3;
  RDODE0 = sqrt( (xOD0 - xE).^2 + (yOD0 - yE).^2 );

  X0 = [ thD0', xD0', yD0', xOD0', yOD0', RDODE0'];
  X = fsolve( @(x) BarrierEndSystem( x(:,1),x(:,2),x(:,3),x(:,4)
    ,x(:,5),x(:,6) ), X0, options);

  xOD = X(:,4)';
  yOD = X(:,5)';
  xD = X(:,2)';
  yD = X(:,3)';
  RD = hypot(xD, yD);
  teD = atan2(yD, xD);
end

%% Flux-barrier points
% We already have the potentials of the two flux-barrier
    sidelines
phiA = phi_fluid( rho_map(RA), xi_map(teA), rho0);
phiB = phi_fluid( rho_map(RB), xi_map(teB), rho0);

phiC = phi_fluid( rho_map(RC), xi_map(teC), rho0);
phiD = phi_fluid( rho_map(RD), xi_map(teD), rho0);

%% Code for single Nstep
% dphiAC = (phiC - phiAprime)./Nstep;
% dphiBD = (phiD - phiBprime)./Nstep;
%
% % we create the matrix of potentials phi needed for
   points intersections
% PhiAC = phiAprime + cumsum( repmat(dphiAC, Nstep - 1,
   1) );
% PhiBD = phiBprime + cumsum( repmat(dphiBD, Nstep - 1,
   1) );
%
% PhiAC_vec = reshape(PhiAC, numel(PhiAC), 1);
% PhiBD_vec = reshape(PhiBD, numel(PhiBD), 1);
% PsiAC_vec = reshape( repmat( psiA, Nstep-1, 1), numel(
   PhiAC), 1 );
```

```
% PsiBD_vec = reshape( repmat( psiB, Nstep-1, 1), numel( PhiBD), 1 );
%
% % we find all the barrier points along the streamline
% PsiPhi = @(rho,xi, psi,phi, rho0) ...
%   [psi - psi_fluid(rho, xi, rho0)
%    phi - phi_fluid(rho, xi, rho0)];
%
% X0 = [repmat(rho0*1.1, numel(PhiAC_vec), 1), repmat(pi/4, numel(PhiAC_vec), 1)];
% RhoXi_AC = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiAC_vec, PhiAC_vec, rho0 ), X0, options);
% RhoXi_BD = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiBD_vec, PhiBD_vec, rho0 ), X0, options);
%
% R_AC = reshape( r_map(RhoXi_AC(:,1)), Nstep-1, Nb );
% te_AC = reshape( th_map(RhoXi_AC(:,2)), Nstep-1, Nb );
% R_BD = reshape( r_map(RhoXi_BD(:,1)), Nstep-1, Nb );
% te_BD = reshape( th_map(RhoXi_BD(:,2)), Nstep-1, Nb );

%% Code for different Nsteps
% we find all the barrier points along the streamline
PsiPhi = @(rho,xi, psi,phi, rho0) ...
  [psi - psi_fluid(rho, xi, rho0)
  phi - phi_fluid(rho, xi, rho0)];

% barrier(Nb).R_AC = 0;
% barrier(Nb).R_BD = 0;
% barrier(Nb).te_AC = 0;
% barrier(Nb).te_BD = 0;
barrier(Nb) = struct;

for bkk = 1:Nb
  dphiAC = (phiC(bkk) - phiA(bkk))./Nstep(bkk);
  dphiBD = (phiD(bkk) - phiB(bkk))./Nstep(bkk);
  % we create the matrix of potentials phi needed for points intersections
  PhiAC = phiA(bkk) + cumsum( repmat(dphiAC', Nstep(bkk) - 1, 1) );
  PhiBD = phiB(bkk) + cumsum( repmat(dphiBD', Nstep(bkk) - 1, 1) );
```

```
  PsiAC = repmat( psiA(bkk), Nstep(bkk)-1, 1);
  PsiBD = repmat( psiB(bkk), Nstep(bkk)-1, 1);

% 1st try
%   X0 = [repmat(rho0*1.1, numel(PhiAC), 1), repmat(pi
    /4, numel(PhiAC), 1)];
% 2nd try
%   X0 = [repmat(rho0*1.1, numel(PhiAC), 1), repmat(
    xi_map(teE(bkk)), numel(PhiAC), 1)];
% 3rd try
  X0 = [linspace(rho0, Dend/2, numel(PhiAC))', linspace(pi/4,
    xi_map(teE(bkk)), numel(PhiAC))'];
  RhoXi_AC = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiAC, PhiAC,
    rho0 ), X0, options);
  RhoXi_BD = fsolve( @(x) PsiPhi( x(:,1),x(:,2), PsiBD, PhiBD,
    rho0 ), X0, options);

  R_AC = r_map(RhoXi_AC(:,1));
  te_AC = th_map(RhoXi_AC(:,2));
  R_BD = r_map(RhoXi_BD(:,1));
  te_BD = th_map(RhoXi_BD(:,2));

  if deb
    barrier(bkk).R_AC = R_AC/ScalingFactor;
    barrier(bkk).R_BD = R_BD/ScalingFactor;
    barrier(bkk).te_AC = te_AC;
    barrier(bkk).te_BD = te_BD;
  end

  % output of points
%   barrier(bkk).Zeta = [...
  Zeta = [...
    % top side
    xE(bkk) + 1j*yE(bkk)
    xOC(bkk) + 1j*yOC(bkk)
    xC(bkk) + 1j*yC(bkk)
    flipud( R_AC.*exp(1j*te_AC) )
    xA(bkk) + 1j*yA(bkk)
    % bottom side
    xB(bkk) + 1j*yB(bkk)
    R_BD.*exp(1j*te_BD)
```

```
      xD(bkk) + 1j*yD(bkk)
      xOD(bkk) + 1j*yOD(bkk)
      xE(bkk) + 1j*yE(bkk)
      ]/ScalingFactor;

    barrier(bkk).X = real(Zeta);
    barrier(bkk).Y = imag(Zeta);

    % magnet central base point
    barrier(bkk).Rm = Rmag(bkk)/ScalingFactor;

    barrier(bkk).barrier_angles_el = barrier_angles_el(bkk);

  end

  %% plot
  if deb

    % draw the rotor
    figure
    hold on
    tt = linspace(0,pi/p,50);
    plot(R0/ScalingFactor*cos(tt), R0/ScalingFactor*sin(tt), 'k');
    plot(Dr/2/ScalingFactor*cos(tt), Dr/2/ScalingFactor*sin(tt), 'k');
    axis equal
    % plot the flux-barrier central point
    plot(RA/ScalingFactor.*exp(1j*teA), 'rd')
    plot(RB/ScalingFactor.*exp(1j*teB), 'bo')

    plot(xE/ScalingFactor, yE/ScalingFactor,'ko')

    plot(xOC/ScalingFactor, yOC/ScalingFactor,'go')
    plot(xC/ScalingFactor, yC/ScalingFactor,'ro')
    plot(xOD/ScalingFactor, yOD/ScalingFactor,'co')
    plot(xD/ScalingFactor, yD/ScalingFactor,'bo')

    %
    % plot(R_AC.*exp(j*te_AC),'r.-')
    % plot(R_BD.*exp(j*te_BD),'b.-')
```

```
    for bkk = 1:Nb
      % plot flux-barrier sideline points
      plot(barrier(bkk).R_AC.*exp(1j*barrier(bkk).te_AC),'r.-')
      plot(barrier(bkk).R_BD.*exp(1j*barrier(bkk).te_BD),'b.-')

      % plot all the complete flux-barrier
      plot(barrier(bkk).X, barrier(bkk).Y, '.-')
    end
    pause(1e-3)

  end

  end
```

5.3 Draw fluid barrier

```
function draw_fluid_barrier(b)

for bkk = 1:length(b)
  xE = b(bkk).X(1);
  yE = b(bkk).Y(1);
  xEOC = b(bkk).X(2);
  yEOC = b(bkk).Y(2);
  xC = b(bkk).X(3);
  yC = b(bkk).Y(3);

  xD = b(bkk).X(end-2);
  yD = b(bkk).Y(end-2);
  xDOE = b(bkk).X(end-1);
  yDOE = b(bkk).Y(end-1);

  X = b(bkk).X(3:end-2);
  Y = b(bkk).Y(3:end-2);

  mi_drawpolyline([X, Y])

  if xEOC == xC && yEOC == yC
    mi_drawline(xE,yE, xC,yC)
  else
    mi_draw_arc(xE,yE, xEOC,yEOC, xC,yC, 1)
  end

  if xDOE == xD && yDOE == yD
    mi_drawline(xD,yD, xE,yE)
  else
    mi_draw_arc(xD,yD, xDOE,yDOE, xE,yE, 1)
  end

end

end
```

6 Python Code

6.1 Main file

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  5 21:31:31 2018

@author: Giacomo
"""

from fluid_functions import *

deb = 0;

## DATA
rotor = structtype();
rotor.p = 2; # number of pole pairs
mm = 1e-3; # millimeters
rotor.De = 200*mm; # [m], rotor outer diameter

rotor.Nb = 3; # number of flux-barriers
rotor.tb = np.array([4, 8, 15])*mm; # flux-barrier widths
rotor.wc = np.array([3, 7, 12, 10])*mm; # flux-carrier widths
rotor.Nstep = np.array([2, 4, 6]); # number of steps to draw the flux-barrier side
rotor.wrib_t = 1*mm; # [m], tangential iron rib width

# you can input flux-barrier angles or let the program compute them
#rotor.barrier_angles_el = np.array([14,26,38])*2; # [deg], electrical flux-barrier angles

# or you can also input a factor to reach the top or the bottom of each barrier
# (do not exceed 100%)
#rotor.barrier_end_wf = np.array([20,50,80])/100; # flux-barrier-end weight factors

#rotor.barrier_end = 'rect'; # choose 'rect' or comment

# you can define the rib width or comment
rotor.wrib = np.array([1,2,4])*mm; # [m], radial iron rib widths
# You can define the magnet width or comment
#rotor.wm = np.array([10,20,40])*mm;
```

```
## barrier points computation
barrier = structtype();
barrier = calc_fluid_barrier(rotor, deb);



## simple matlab plot
for bkk in range(0,np.size(barrier.X)):
    plt.plot(np.squeeze(barrier.X[bkk]), np.squeeze(barrier.Y[bkk]), '.-')

plt.axis('equal')
plt.show()
```

6.2 Fluid functions

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Apr  5 21:49:54 2018

@author: Giacomo
"""

import numpy as np
import scipy as sp
from matplotlib import pyplot as plt

# I define a dummy class for Matlab structure like objects
class structtype():
    pass


def psi_fluid(rho,xi,rho0):
    return (rho**2 - rho0**2)/rho*np.sin(xi);
#    return (np.power(rho,2) - rho0**2)/rho*np.sin(xi);

def phi_fluid(rho,xi,rho0):
    return (np.power(rho,2) + rho0**2)/rho*np.cos(xi);

def xi_fluid(psi,rho,rho0):
    return np.arcsin(psi*rho/(np.power(rho,2) - rho0**2));

def rho_fluid(psi,xi,rho0):
    return ( psi + np.sqrt(np.power(psi,2) + 4*np.power(np.sin(xi),2)*rho0
                                    **2) )/(2*np.sin(xi));

def r_map(rho):
    return np.power(rho, 1/p);

def th_map(xi):
    return xi/p;

def rho_map(r):
    return np.power(r, p);

def xi_map(th):
    return th*p;

def vr(r,th,R0):
```

```
    return p*(np.power(r,(p-1)) - R0**(2*p)/np.power(r,(p+1)))*np.cos(p*th);

def vt(r,th,R0):
    return -p*( np.power(r,(p-1)) + R0**(2*p)/np.power(r,(p+1)) )*np.sin(p*th);

def vx(vr_v,vth_v,th):
    return vr_v*np.cos(th) - vth_v*np.sin(th);

def vy(vr_v,vth_v,th):
    return vr_v*np.sin(th) + vth_v*np.cos(th);


def CentralPt_Eq(th, *args):
    psiCentralPt, rho0, mCentral, qCentral = args;
    return np.multiply( r_map( rho_fluid(psiCentralPt, xi_map(th), rho0) ),
                  ( np.sin(th) - mCentral*np.cos(th) ) ) - qCentral;


def BarrierEndSystem(X, *args):
#    th,xd,yd,xo,yo,R = X;
    th = X[0:Nb];
    xd = X[1*Nb:2*Nb];
    yd = X[2*Nb:3*Nb];
    xo = X[3*Nb:4*Nb];
    yo = X[4*Nb:5*Nb];
    R  = X[5*Nb:6*Nb];

    psiA, rho0, xE, yE = args;
    R0 = r_map(rho0);
    firstEq = xd - np.multiply( r_map(rho_fluid(psiA, p*th, rho0)), np.cos(th) );
    seconEq = yd - np.multiply( r_map(rho_fluid(psiA, p*th, rho0)), np.sin(th) );
    thirdEq = (xd - xo)**2 + (yd - yo)**2 - R**2;
#    thirdEq = (xE - xo)**2 + (yE - yo)**2 - R**2;
    fourtEq = (xE - xo)**2 + (yE - yo)**2 - R**2;
    fifthEq = np.multiply( (xo - xd), vx( vr( r_map(rho_fluid(psiA, p*th, rho0)),th,R0 ), vt( r_map(rho_fluid(psiA, p*th, rho0)) ,th,R0 ), th) ) + np.multiply( (yo - yd), vy( vr( r_map(rho_fluid(psiA, p*th, rho0)),th,R0 ), vt( r_map(rho_fluid(psiA, p*th,
```

```
                                             rho0)) ,th,R0 ), th) );
    sixthEq = np.multiply(xo - xE,yE) - np.multiply(yo - yE,xE);
    return np.concatenate([firstEq,
                           seconEq,
                           thirdEq,
                           fourtEq,
                           fifthEq,
                           sixthEq])

def PsiPhi(X, *args):
    psi, phi, rho0, N = args;
    rho  = X[0:N];
    xi   = X[N:2*N];
    return np.concatenate( [psi - psi_fluid(rho, xi, rho0),
            phi - phi_fluid(rho, xi, rho0)] )


# MAIN function definition
def calc_fluid_barrier(r, deb):
    "CALC_FLUID_BARRIER computes the flux-barrier points along the
                                            streamline function."

    Dr = r.De; # [m], rotor outer diameter
    ScalingFactor = 1/( 10**(round(np.log10(Dr))) );
    # ScalingFactor = 1;
    Dr = Dr*ScalingFactor;

    pi = np.pi;
    global p, Nb # I have been lazy here...
    p = r.p; # number of pole pairs
    Nb = r.Nb; # number of flux-barriers
    tb = r.tb*ScalingFactor; # flux-barrier widths
    wc = r.wc*ScalingFactor; # flux-carrier widths
    Nstep = r.Nstep; # number of steps to draw the flux-barrier side

    wrib_t = r.wrib_t*ScalingFactor; # [m], tangential iron rib width


    if hasattr(r,'barrier_angles_el'):
        barrier_angles_el = r.barrier_angles_el; # [deg], electrical flux-
                                            barrier angles
        AutoBarrierEndCalc = 0;
    else:
        barrier_angles_el =  np.zeros(Nb);
        AutoBarrierEndCalc = 1;
```

```
        if hasattr(r,'barrier_end_wf'):
            wf = r.barrier_end_wf;
        else:
            wf = 0.5*np.ones(Nb);

    if hasattr(r,'wm'):
        wm = r.wm*ScalingFactor;
    else:
        wm = 0;

    if hasattr(r,'wrib'):
        wrib = r.wrib*ScalingFactor + wm; # [m], radial iron rib widths
    else:
        wrib = np.zeros([1,Nb]) + wm;

    Dend = Dr - 2*wrib_t; # [m], flux-barrier end diameter
    Dsh = Dend - 2*( np.sum(tb) + np.sum(wc) ); # [m], shaft diameter
    R0 = Dsh/2; # [m], shaft radius
    barrier_angles = barrier_angles_el/p; # [deg], flux-barrier angles
    if hasattr(r,'barrier_end'):
        barrier_end = r.barrier_end;
    else:
        barrier_end = '';

    ## Precomputations
    rho0 = rho_map(R0);

    ## Central base points
    RAprime = Dend/2 - np.concatenate( (np.array([0]), np.cumsum( tb[0:-1] ) ) ) - np.cumsum( wc[0:-1] ); # top
    RBprime = RAprime - tb; # bottom
    te_qAxis = pi/(2*p); # q-axis angle in rotor reference frame

    # get A' and B' considering rib and magnet widths
    mCentral = np.tan(te_qAxis); # slope
    qCentral = np.tile( -wrib/2/np.cos(te_qAxis), 2); # intercept

    psiCentralPtA = psi_fluid(rho_map(RAprime), xi_map(te_qAxis), rho0);
    psiCentralPtB = psi_fluid(rho_map(RBprime), xi_map(te_qAxis), rho0);
    psiCentralPt = np.array( np.concatenate( (psiCentralPtA, psiCentralPtB ) ) );
    psiA = psiCentralPtA;
    psiB = psiCentralPtB;
```

```
    FunctionTolerance = 10*np.spacing(1);
    StepTolerance = 1e4*np.spacing(1);

    X0 = np.repeat(te_qAxis, 2*Nb);
    data = (psiCentralPt,rho0,mCentral,qCentral);
    # test function
#     print( CentralPt_Eq(X0, *data ) )

    teAB = sp.optimize.fsolve(CentralPt_Eq, X0, args=data, xtol=StepTolerance, epsfcn=FunctionTolerance);
    teA = teAB[0:Nb];
    teB = teAB[Nb:];
    RA = r_map( rho_fluid(psiA, xi_map(teA), rho0) );
    RB = r_map( rho_fluid(psiB, xi_map(teB), rho0) );

    # central base points
    zA = RA*np.exp(1j*teA);
    zB = RB*np.exp(1j*teB);
    xA = zA.real;
    yA = zA.imag;
    xB = zB.real;
    yB = zB.imag;

    # magnet central base point radius computation
    RAsecond = RA*np.cos(te_qAxis - teA);
    RBsecond = RB*np.cos(te_qAxis - teB);

    Rmag = (RAprime + RAsecond + RBprime + RBsecond)/4;

    # 1st test --> OK!
#     print(RA,teA,RB,teB)
#     print(xA,yA,xB,yB)

    # Outer base points C,D preparation
    RCprime = Dend/2;
    teCprime = th_map( xi_fluid(psiA, rho_map(RCprime), rho0) );
    xCprime = Dend/2*np.cos(teCprime);
    yCprime = Dend/2*np.sin(teCprime);

    RDprime = Dend/2;
    teDprime = th_map( xi_fluid(psiB, rho_map(RDprime), rho0) );
    xDprime = Dend/2*np.cos(teDprime);
    yDprime = Dend/2*np.sin(teDprime);
```

```
    if AutoBarrierEndCalc:
        teE = ( teCprime*(1 - wf) + teDprime*wf );
        aphE = pi/2/p - teE;
        barrier_angles = 180/np.pi*aphE;
        barrier_angles_el = p*barrier_angles;
    else:
        aphE = barrier_angles*pi/180;
        teE = pi/2/p - aphE;

    xE = Dend/2*np.cos(teE);
    yE = Dend/2*np.sin(teE);

    # 2nd test --> OK!
#     print(xE,yE)



    ## Outer base points C (top)
    if barrier_end == 'rect':
        RC = RCprime;
        teC = teCprime;
        xC = xCprime;
        yC = yCprime;
        xOC = xC;
        yOC = yC;

    else:
        # 1st try
#         X0 = [ 1.5*teE, 0.9*xE, 0.9*yE, 0.8*xE, 0.8*yE, 0.25*xE];
        # best try
        xC0 = ( xE + xCprime + 0.1*xA )/(2 + 0.1);
        yC0 = ( yE + yCprime )/2;
        thC0 = np.arctan(yC0/xC0);
        xOC0 = ( xE + xC0 + 0 )/3;
        yOC0 = ( yE + yC0 + 0 )/3;
        RCOCE0 = np.sqrt( (xOC0 - xE)**2 + (yOC0 - yE)**2 );

        X0 = [ thC0, xC0, yC0, xOC0, yOC0, RCOCE0];
        X0 = np.reshape(X0, Nb*6);

        data = (psiA, rho0, xE, yE);
        X = sp.optimize.fsolve( BarrierEndSystem, X0, args=data);

        xOC = X[3*Nb:4*Nb];
        yOC = X[4*Nb:5*Nb];
```

```
        xC = X[1*Nb:2*Nb];
        yC = X[2*Nb:3*Nb];
        RC = np.sqrt(xC**2 + yC**2);
        teC = np.arctan2(yC, xC);

    # 3rd test --> OK!
#    print(xOC)
#    print(yOC)
#    print(xC)
#    print(yC)
#    print(RC)
#    print(teC)


    ## Outer base points D (bottom)
    if barrier_end == 'rect':
        RD = RDprime;
        teD = teDprime;
        xD = xDprime;
        yD = yDprime;
        xOD = xD;
        yOD = yD;

    else:
        # 1st try
#        X0 = [ 0.8*teE, 0.8*xE, 0.8*yE, 0.9*xE, 0.9*yE, 0.2*xE];
        # best try
        xD0 = ( xE + xDprime )/2;
        yD0 = ( yE + yDprime )/2;
        thD0 = np.arctan(yD0/xD0);
        xOD0 = ( xE + xD0 + xC )/3;
        yOD0 = ( yE + yD0 + yC )/3;
        RDODE0 = np.sqrt( (xOD0 - xE)**2 + (yOD0 - yE)**2 );

        X0 = [ thD0, xD0, yD0, xOD0, yOD0, RDODE0];
        X0 = np.reshape(X0, Nb*6);

        data = (psiB, rho0, xE, yE);
        X = sp.optimize.fsolve( BarrierEndSystem, X0, args=data);

        xOD = X[3*Nb:4*Nb];
        yOD = X[4*Nb:5*Nb];
        xD = X[1*Nb:2*Nb];
        yD = X[2*Nb:3*Nb];
        RD = np.sqrt(xD**2 + yD**2);
```

```
        teD = np.arctan2(yD, xD);

    # 4th test --> OK!
#     print(xOD)
#     print(yOD)
#     print(xD)
#     print(yD)
#     print(RD)
#     print(teD)


    ## Flux-barrier points
    # We already have the potentials of the two flux-barrier sidelines
    phiA = phi_fluid( rho_map(RA), xi_map(teA), rho0);
    phiB = phi_fluid( rho_map(RB), xi_map(teB), rho0);

    phiC = phi_fluid( rho_map(RC), xi_map(teC), rho0);
    phiD = phi_fluid( rho_map(RD), xi_map(teD), rho0);

    barrier = structtype();




    XX = [];
    YY = [];
    Rm = [];

    for bkk in range(0,Nb):
        dphiAC = np.divide(phiC[bkk] - phiA[bkk], Nstep[bkk]);
        dphiBD = np.divide(phiD[bkk] - phiB[bkk], Nstep[bkk]);
        # we create the matrix of potentials phi needed for points intersections
        PhiAC = phiA[bkk] + np.cumsum( np.tile(dphiAC, Nstep[bkk] - 1) );
        PhiBD = phiB[bkk] + np.cumsum( np.tile(dphiBD, Nstep[bkk] - 1) );
        PsiAC = np.tile( psiA[bkk], Nstep[bkk]-1);
        PsiBD = np.tile( psiB[bkk], Nstep[bkk]-1);

        X0 = np.concatenate( [np.linspace(rho0, Dend/2, np.size(PhiAC)),
                              np.linspace(pi/4, xi_map(teE[bkk]),
                              np.size(PhiAC))] );

        data = (PsiAC, PhiAC, rho0, Nstep[bkk]-1);
        RhoXi_AC = sp.optimize.fsolve( PsiPhi, X0, args=data );
```

```
        data = (PsiBD, PhiBD, rho0, Nstep[bkk]-1);
        RhoXi_BD = sp.optimize.fsolve( PsiPhi, X0, args=data );

        R_AC = r_map( RhoXi_AC[0:Nstep[bkk]-1] );
        te_AC = th_map( RhoXi_AC[Nstep[bkk]-1:] );
        R_BD = r_map( RhoXi_BD[0:Nstep[bkk]-1] );
        te_BD = th_map( RhoXi_BD[Nstep[bkk]-1:] );

        # 5th test --> OK!
#        print(R_AC, te_AC)
#        print(R_BD, te_BD)

        Zeta = np.concatenate( [[
                # top side
                xE[bkk] + 1j*yE[bkk],
                xOC[bkk] + 1j*yOC[bkk],
                xC[bkk] + 1j*yC[bkk] ],
                np.flipud( np.multiply(R_AC, np.exp(1j*te_AC)) ),
                [xA[bkk] + 1j*yA[bkk],
                 # bottom  side
                 xB[bkk] + 1j*yB[bkk]],
                np.multiply( R_BD, np.exp(1j*te_BD) ),
                [xD[bkk] + 1j*yD[bkk],
                 xOD[bkk] + 1j*yOD[bkk],
                 xE[bkk] + 1j*yE[bkk]]
        ] )/ScalingFactor;

        X = Zeta.real;
        Y = Zeta.imag;

        XX.append([X]);
        YY.append([Y]);

        # magnet central base point
        Rm.append(Rmag[bkk]/ScalingFactor);

    barrier.X = XX;
    barrier.Y = YY;
    barrier.Rm = Rm;


    if deb == 1:
        Apt = np.multiply(RA/ScalingFactor, np.exp(1j*teA) );
        Bpt = np.multiply(RB/ScalingFactor, np.exp(1j*teB) );
        Cpt = np.multiply(RC/ScalingFactor, np.exp(1j*teC) );
```

```
        Dpt = np.multiply(RD/ScalingFactor, np.exp(1j*teD) );
        plt.plot( [Apt.real,Cpt.real], [Apt.imag,Cpt.imag], "ro" )
        plt.plot( [Bpt.real,Dpt.real], [Bpt.imag,Dpt.imag], "bo" )
        plt.plot( xE/ScalingFactor, yE/ScalingFactor, "ko" )

        for bkk in range(0,Nb):
            plt.plot(np.squeeze(barrier.X[bkk]), np.squeeze(barrier.Y[bkk]
                                ))

        plt.axis('equal')
        plt.show()



    return barrier
```